# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

- Computadores operam em sequências de bits (ou seja, sequências que consistem apenas nos símbolos 0 e 1).
- Portanto, são necessários esquemas de codificação que peguem o texto escrito em alfabetos mais ricos (como os alfabetos que sustentam as línguas humanas) e convertam esse texto em longos cadeias de bits.
- A maneira mais simples de fazer isso seria usar um número fixo de bits para cada símbolo do alfabeto e, em seguida, apenas concatenar as cadeias de bits de cada símbolo para formar o texto.
- Para dar um exemplo básico, suponha que queremos codificar as 26 letras do inglês, mais o espaço (para separar as palavras) e cinco caracteres de pontuação: vírgula, ponto, ponto de interrogação, ponto de exclamação e apóstrofo.
- Isso nos daria 32 símbolos no total a serem codificados.
- Podemos formar $2^b$ sequências diferentes de $b$ bits e, portanto, se usarmos 5 bits por símbolo, poderemos codificar $2^b = 2^5 = 32$ símbolos, o suficiente para nossos propósitos.
- Assim, por exemplo, podemos deixar a string de bits 00000 representar **a**, a string de bits 00001 representar **b**, e assim por diante até 11111, que pode representar o apóstrofo.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

- Observe que o mapeamento de cadeias de bits para símbolos é arbitrário.
- Esquemas de codificação como ASCII funcionam exatamente dessa maneira, exceto que eles usam um número maior de bits por símbolo para lidar com conjuntos de caracteres maiores, incluindo letras maiúsculas, parênteses e todos os outros símbolos especiais que você vê em uma máquina de escrever ou computador teclado.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

- Vamos pensar em nosso exemplo básico com apenas 32 símbolos.
- **Existe algo mais que poderíamos pedir de um esquema de codificação? Poderíamos ser mais eficientes?**
- Não poderíamos pedir para codificar cada símbolo usando apenas quatro bits, já que $2^b = 2^4 = 16$? Porém, não é suficiente para o número de símbolos que temos.
- No entanto, não está claro se, em grandes extensões de texto, realmente precisamos gastar uma média de cinco bits por símbolo.
- Se pensarmos bem, as letras na maioria dos alfabetos humanos não são usadas com a mesma frequência.
- Em inglês, por exemplo, as letras **e, t, a, o, i** e **n** são usadas com muito mais frequência do que **q, j, x** e **z**.
- Portanto, é realmente um tremendo desperdício traduzi-los todos no mesmo número de bits!!!
- Poderíamos usar um pequeno número de bits para as letras frequentes e um número maior de bits para as menos frequentes, com o objetivo de acabar usando menos de cinco bits por letra quando calculamos a média de uma longa sequência de texto típico.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

- Essa questão de reduzir o número médio de bits por letra é um problema fundamental na área de compressão de dados.
- Quando arquivos grandes precisam ser enviados por redes de comunicação ou armazenados em discos rígidos, é importante representá-los da forma mais compacta possível!
- Obviamente, desde que um leitor subsequente do arquivo seja capaz de reconstruí-lo corretamente.
- Uma grande quantidade de pesquisa é dedicada ao projeto de algoritmos de compressão que podem receber arquivos como entrada e reduzir seu espaço por meio de esquemas de codificação eficientes.
- Descrevemos agora uma das formas fundamentais de formular esse problema, chegando à questão de como podemos construir a maneira ideal de aproveitar as frequências não uniformes das letras.
- Em certo sentido, tal solução ótima é uma resposta muito atraente para o problema da compressão de dados:

*Ela espreme todos os ganhos disponíveis das não uniformidades nas frequências (ou seja, aproveitando que algumas letras são mais frequentes que outras nos textos).*

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Esquemas de codificação de comprimento**

- Antes da Internet, antes do computador digital, antes do rádio e do telefone, havia o telégrafo.
- A comunicação por telégrafo era muito mais rápida do que as alternativas contemporâneas de entrega manual de mensagens por trem ou a cavalo.
- Mas os telégrafos só eram capazes de transmitir pulsos por um fio e, portanto, se você quisesse enviar uma mensagem, precisava de uma maneira de codificar o texto de sua mensagem como uma sequência de pulsos.
- Para lidar com essa questão, o pioneiro da comunicação telegráfica, Samuel Morse, desenvolveu o código Morse, traduzindo cada letra em uma sequência de pontos (pulsos curtos) e traços (pulsos longos).
- Para nossos propósitos, podemos pensar em pontos e traços como zeros e uns, e isso é simplesmente um mapeamento de símbolos em cadeias de bits, assim como em ASCII.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Esquemas de codificação de comprimento**

- Morse entendeu que alguém poderia se comunicar de forma mais eficiente codificando letras frequentes com strings curtas e, portanto, essa foi a abordagem que ele adotou.
- Ele consultou as gráficas locais para obter estimativas de frequência para as letras em inglês.
- Assim, o código Morse mapeia ***E*** para 0 (um único ponto), ***T*** para 1 (um único traço), ***A*** para 01 (ponto-traço) e em geral, mapeia letras mais frequentes para cadeias de bits mais curtas.
- Na verdade, o código Morse usa strings tão curtas para as letras que a codificação das palavras se torna ambígua.
- Por exemplo, apenas usando o que sabemos sobre a codificação de ***E***, ***T*** e ***A***, vemos que a string 0101 pode corresponder a qualquer uma das sequências de letras:
  - ✓ ETA
  - ✓ AA
  - ✓ ETET
  - ✓ AET
  - ✓ Combinações com outras letras, por exemplo RT.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Esquemas de codificação de comprimento**

- Para lidar com essa ambigüidade, as transmissões do código Morse envolvem pausas curtas entre as letras (portanto, a codificação de ***AA*** seria, na verdade, ponto-traço-pausa-ponto-traço-pausa).
- Esta é uma solução razoável, usando cadeias de bits muito curtas e depois introduzindo pausas.
- Porém, significa que não codificamos as letras usando apenas 0 e 1; na verdade, nós o codificamos usando um alfabeto de três letras de 0, 1 e "pausa".
- Assim, se realmente precisássemos codificar tudo usando apenas os bits 0 e 1, seria necessário haver alguma codificação adicional na qual a pausa fosse mapeada para bits.
- Como resolver o problema da ambiguidade sem usar pausas?

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

## Códigos de prefixo

- O problema de ambigüidade no código Morse surge porque existem pares de letras onde a cadeia de bits que codifica uma letra é **um prefixo da cadeia de bits que codifica outra.**
- Para eliminar esse problema e, portanto, obter um esquema de codificação que tenha uma interpretação bem definida para cada sequência de bits, basta mapear letras para cadeias de bits de forma **que nenhuma codificação seja um prefixo de qualquer outra**.
- Dizemos que um código de prefixo para um conjunto $S$ de letras é uma função $\gamma$ que mapeia cada letra $x \in S$ para alguma sequência de zeros e uns, de tal forma que para $x$ distintos, $y \in S$, a sequência $\gamma(x)$ não é um prefixo da sequência $\gamma(y)$.
- Agora suponha que temos um texto que consiste em uma sequência de letras:

$$x_1 x_2 x_3 \ldots x_n.$$

- Podemos converter isso em uma sequência de bits simplesmente codificando cada letra como uma sequência de bits usando $\gamma$.
- Em seguida, concatenando todas essas sequências de bits juntas:

$$\gamma(x_1)\gamma(x_2)\gamma(x_3)\ldots\gamma(x_n).$$

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Códigos de prefixo**

- Se então entregarmos esta mensagem a um destinatário que conheça a função $\gamma$ , ele poderá reconstruir o texto de acordo com a seguinte regra:
  1. Examine a sequência de bits da esquerda para a direita.
  2. Assim que você tiver visto bits suficientes para corresponder à codificação de alguma letra, imprima isso como a primeira letra do texto.
  3. Esta deve ser a primeira letra correta, já que nenhum prefixo mais curto ou mais longo da sequência de bits poderia codificar qualquer outra letra.
  4. Agora exclua o conjunto de bits correspondente da frente da mensagem e itere.
  5. Desta forma, o destinatário pode produzir o conjunto correto de letras sem que tenhamos que recorrer a artifícios artificiais como pausas para separar as letras.
- Por exemplo, suponha que estamos tentando codificar o conjunto de cinco letras $S = \{a, b, c, d, e\}$. A codificação $\gamma_1$ especificada por

$$\gamma_1(a) = 11, \gamma_1(b) = 01, \gamma_1(c) = 001, \gamma_1(d) = 10, \gamma_1(e) = 000$$

- é um código de prefixo, pois podemos verificar que nenhuma codificação é um prefixo de qualquer outra.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Códigos de prefixo**

- Por exemplo, suponha que estamos tentando codificar o conjunto de cinco letras $S = \{a, b, c, d, e\}$. A codificação $\gamma_1$ especificada por

$$\gamma_1(a) = 11$$
$$\gamma_1(b) = 01$$
$$\gamma_1(c) = 001$$
$$\gamma_1(d) = 10$$
$$\gamma_1(e) = 000$$

- é um código de prefixo, pois podemos verificar que nenhuma codificação é um prefixo de qualquer outra.
- Agora, por exemplo, a string **cecab** seria codificada como **0010000011101**.
- Um destinatário dessa mensagem, sabendo $\gamma_1$, começaria a ler da esquerda para a direita.
- Nem 0 nem 00 codificam uma letra, mas 001 sim, então o destinatário conclui que a primeira letra é **c**.
- Esta é uma decisão segura, já que nenhuma sequência de bits começando com 001 poderia codificar uma letra diferente.
- O destinatário agora itera no restante da mensagem, **0000011101**;
- Em seguida, eles concluirão que a segunda letra é **e**, codificada como **000**.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Códigos de prefixo ótimos**

- Fizemos tudo isso porque algumas letras são mais frequentes do que outras e queremos aproveitar o fato de que letras mais frequentes podem ter codificações mais curtas.
- Para tornar esse objetivo preciso, introduzimos agora alguma notação para expressar as frequências das letras.
- Suponha que para cada letra $x \in S$, exista uma frequência $f_x$, representando a fração de letras no texto que são iguais a $x$.
- Em outras palavras, assumindo existem $n$ letras no total, $nf_x$ dessas letras são iguais a $x$.
- Notamos que a soma das frequências é igual a 1, ou seja:

$$\sum_{x \in S} f_x = 1$$

- **Agora, se usarmos um código de prefixo $\gamma$ para codificar o texto fornecido, qual é o comprimento total de nossa codificação?**
- Isso é simplesmente a soma, sobre todas as letras $x \in S$, do número de vezes que $x$ ocorre vezes o comprimento da cadeia de bits $\gamma(x)$ usada para codificar $x$.
- Usando $|\gamma(x)|$ para denotar o $\gamma(x)$ , podemos escrever isso como comprimento de codificação $(EL)$

$$EL = \sum_{x \in S} nf_x \, |\gamma(x)| = n \sum_{x \in S} f_x \, |\gamma(x)|$$

- Eliminar o coeficiente principal de $n$ da expressão final nos dá $\sum_{x \in S} f_x \, |\gamma(x)|$ , o número médio de bits necessários por letra.
- Nós denotamos essa quantidade como:

$$ABL(\gamma) = \sum_{x \in S} f_x \, |\gamma(x)|$$

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Códigos de prefixo ótimos**

- Para continuar o exemplo anterior, suponha que temos um texto com as letras $S = \{a, b, c, d, e\}$, e suas frequências são as seguintes:

$$f_a = 0.32, f_b = 0.25, f_c = 0.20, f_d = 0.18, f_e = 0.05$$

- Em seguida, o número médio de bits por letra usando o código de prefixo $\gamma_1$ definido anteriormente é

$$ABL(\gamma_1) = \sum_{x \in S} f_x \, |\gamma_1(x)|$$
$$= 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3$$
$$= 2.25$$

- É interessante comparar isso com o número médio de bits por letra usando uma codificação de comprimento fixo.
- Observe que uma codificação de comprimento fixo é um código de prefixo: se todas as letras tiverem codificações do mesmo comprimento, então claramente nenhuma codificação pode ser um prefixo de qualquer outra.
- Com um conjunto $S$ de cinco letras, precisaríamos de três bits por letra para uma codificação de comprimento fixo, já que dois bits só poderiam codificar quatro letras.
- Assim, usar o código $\gamma_1$ reduz os bits por letra de 3 para 2.25, uma economia de 25%.
- E, de fato, $\gamma_1$ não é o melhor que podemos fazer neste exemplo.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Códigos de prefixo ótimos**

- Considere o código de prefixo $\gamma_2$ dado por:

$$\gamma_2(a) = 11$$
$$\gamma_2(b) = 10$$
$$\gamma_2(c) = 01$$
$$\gamma_2(d) = 001$$
$$\gamma_2(e) = 000$$

- O número médio de bits por letra usando $\gamma_2$ é

$$ABL(\gamma_2) = \sum_{x \in S} f_x \, |\gamma_2(x)|$$
$$= 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 2 + 0.18 \cdot 3 + 0.05 \cdot 3$$
$$= 2.23$$

- Reduzimos a necessidade média de bits de 2.25 para 2.23!
- Portanto, agora é natural formular a questão subjacente.

- **Dado um alfabeto e um conjunto de frequências para as letras, gostaríamos de produzir um prefixo código que seja o mais eficiente possível?**
- Ou seja, um código de prefixo que minimize o número médio de bits por letra $ABL(\gamma) = \sum_{x \in S} f_x \, |\gamma(x)|$?
- Se existir, chamaremos tal código de prefixo ótimo!!!

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**O ALGORITMO**

- O espaço de busca para este problema é bastante complicado;
- Inclui todas as formas possíveis de mapear letras para cadeias de bits, sujeitas à propriedade de definição de códigos de prefixo.
- Para alfabetos que consistem em um número extremamente pequeno de letras, é possível pesquisar esse espaço pela força bruta, mas isso rapidamente se torna inviável.
- Agora descrevemos um método guloso para construir um código de prefixo ótimo de forma muito eficiente.
- Como primeiro passo, é útil desenvolver um meio baseado em árvore de representar códigos de prefixo que exponha sua estrutura de forma mais clara do que simplesmente as listas de valores de função que usamos em nossos exemplos anteriores.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

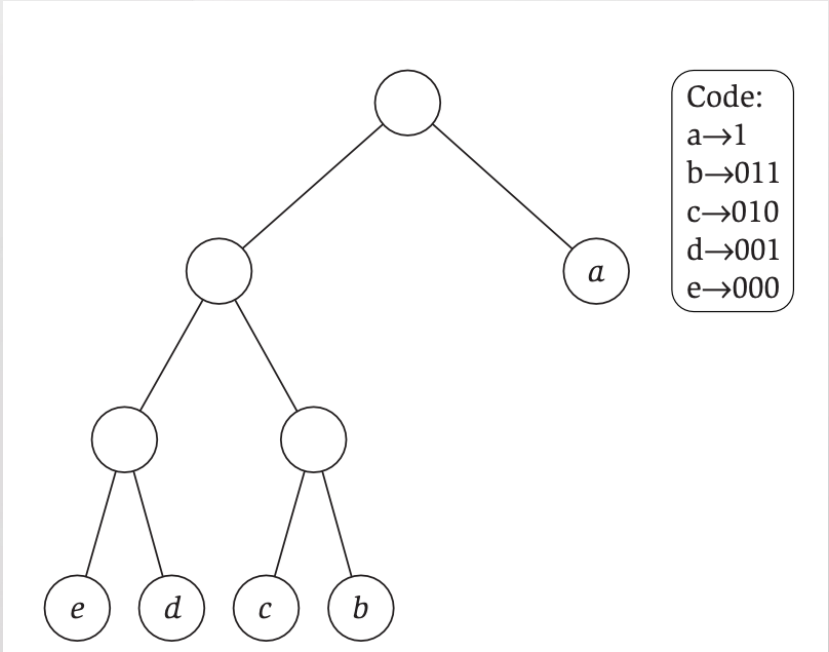
**Representando códigos de prefixo usando árvores binárias**

- Suponha que tomamos uma árvore $T$ na qual cada nó que não é uma folha tem no máximo dois filhos;
- Chamamos essa árvore de **árvore binária**.
- Suponha ainda que o número de folhas seja igual ao tamanho do alfabeto $S$, e rotulamos cada folha com uma letra distinta em $S$.
- Tal árvore binária rotulada $T$ descreve naturalmente um código de prefixo, como segue:
  - Para cada letra $x \in S$, seguimos o caminho da raiz até a folha $x$;
  - cada vez que o caminho vai de um nó para seu filho esquerdo, escrevemos um 0, e cada vez que o caminho vai de um nó para seu filho direito, escrevemos um 1.
  - Tomamos a sequência de bits resultante como a codificação de $x$.

***A.1 A codificação de S construída a partir de T é um código de prefixo.***

**Prova.** Para que a codificação de $x$ seja um prefixo da codificação de $y$, o caminho da raiz até $x$ teria que ser um prefixo do caminho da raiz para $y$.

Mas isso é o mesmo que dizer que $x$ estaria no caminho da raiz para $y$, o que não é possível se $x$ for uma folha.



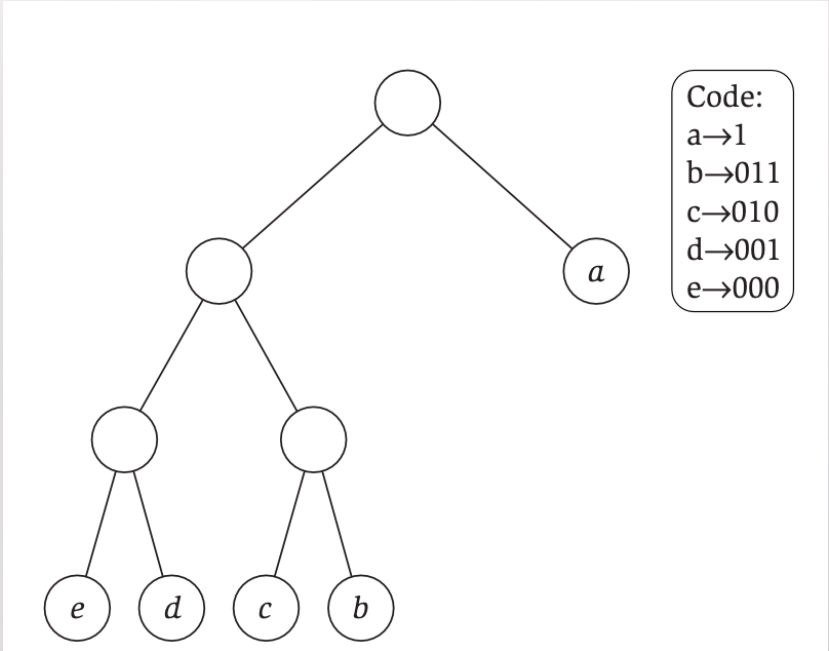
# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Representando códigos de prefixo usando árvores binárias**

- Essa relação entre árvores binárias e códigos de prefixo também funciona na outra direção.
- Dado um código de prefixo $\gamma$ , podemos construir uma árvore binária recursivamente como segue:
  - Começamos com uma raiz;
  - Todas as letras $x \in S$ cujas codificações começam com 0 serão deixadas na subárvore esquerda da raiz, e todas as letras $y \in S$ cujas codificações começam com 1 serão deixadas na subárvore direita da raiz.
  - Agora construímos essas duas subárvores recursivamente usando essa regra.
- Por exemplo, a árvore rotulada na Figura corresponde ao código de prefixo $\gamma_0$ especificado por

$$\gamma_0(a) = 1$$
$$\gamma_0(b) = 011$$
$$\gamma_0(c) = 010$$
$$\gamma_0(d) = 001$$
$$\gamma_0(e) = 000$$

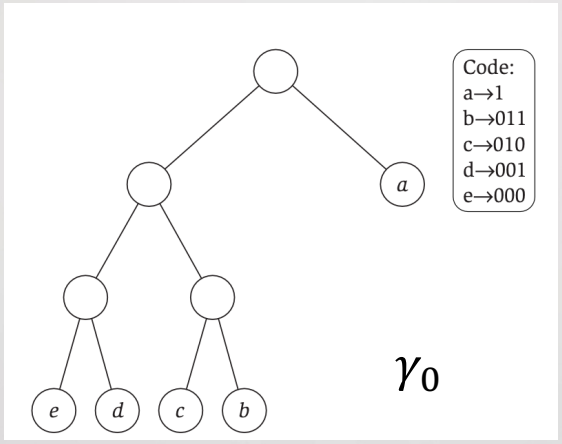
- Para ver isso, observe que a folha rotulada como $\boldsymbol{a}$ é obtida simplesmente retirando a aresta direita da raiz (resultando em uma codificação de 1);
- A folha rotulada $\boldsymbol{e}$ é obtida tomando três arestas esquerdas sucessivas começando da raiz;
- e explicações análogas se aplicam a $\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$.

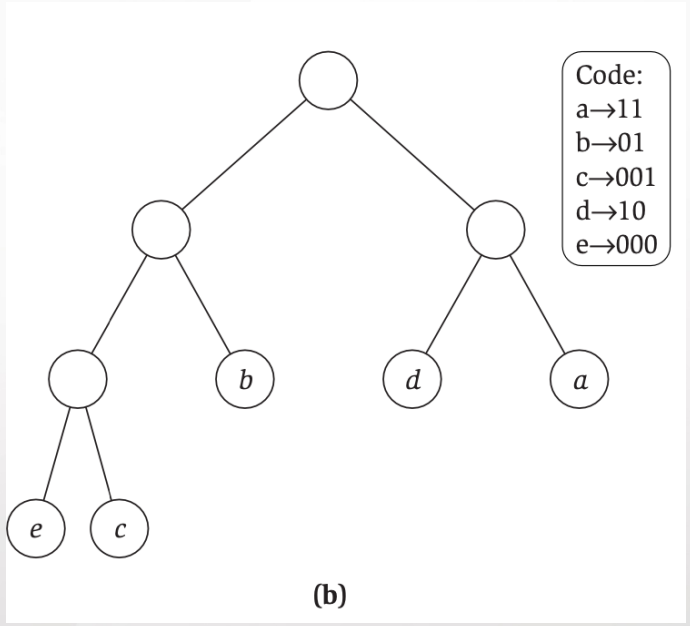


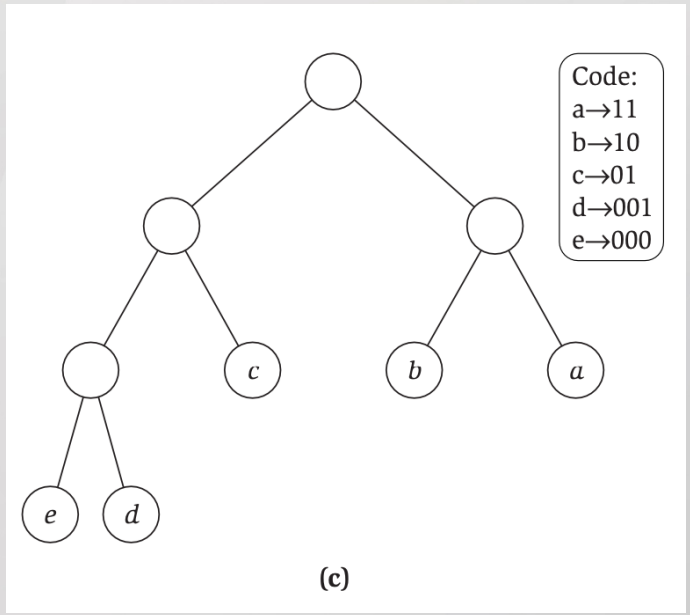
# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Representando códigos de prefixo usando árvores binárias**

- Por raciocínio semelhante, pode-se ver que a árvore rotulada na Figura (b) corresponde ao código de prefixo $\gamma_1$ definido anteriormente.
- E a árvore rotulada na Figura (c) corresponde ao código de prefixo $\gamma_2$ definido anteriormente.
- Observe também que as árvores binárias para os dois códigos de prefixo $\gamma_1$ e $\gamma_2$ são idênticas em estrutura; apenas a rotulagem das folhas é diferente.
- A árvore para $\gamma_0$, por outro lado, tem uma estrutura diferente.



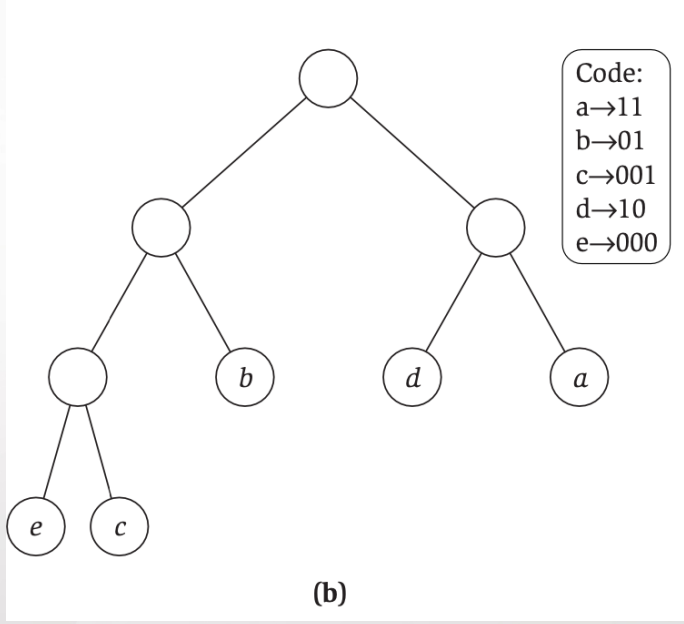
# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Representando códigos de prefixo usando árvores binárias**

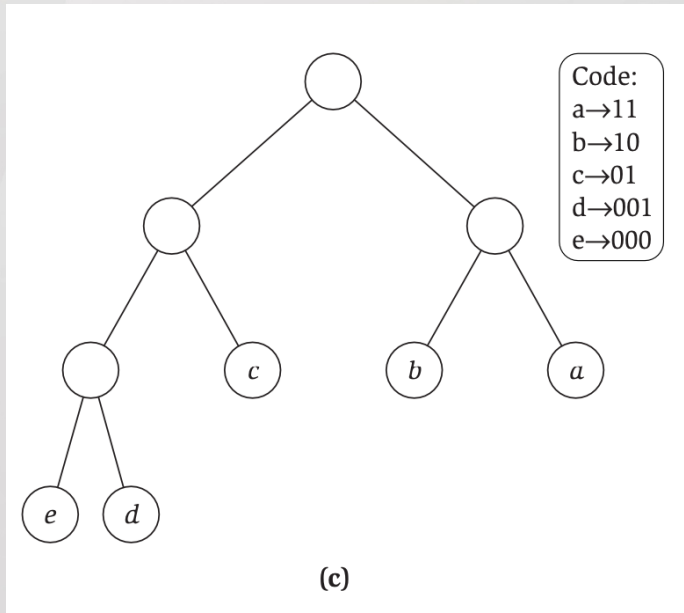
- A busca por um código de prefixo ótimo pode ser vista como a busca por uma árvore binária $T$, juntamente com uma rotulagem das folhas de $T$, que minimiza o número médio de bits por letra.
- Além disso, essa quantidade média tem uma interpretação natural nos termos da estrutura de $T$:

***O comprimento da codificação de uma letra $x \in S$ é simplesmente o comprimento do caminho da raiz até a folha rotulada $x$.***

- Vamos nos referir ao comprimento desse caminho como a profundidade da folha e denotaremos a profundidade de uma folha $v$ em $T$ simplesmente por profundidade $T(v)$.
- Como dois bits de conveniência de notação, descartaremos o subscrito $T$ quando estiver claro no contexto, e frequentemente usaremos uma letra $x \in S$ para também denotar a folha que é rotulada por ele.



(b)



(c)

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

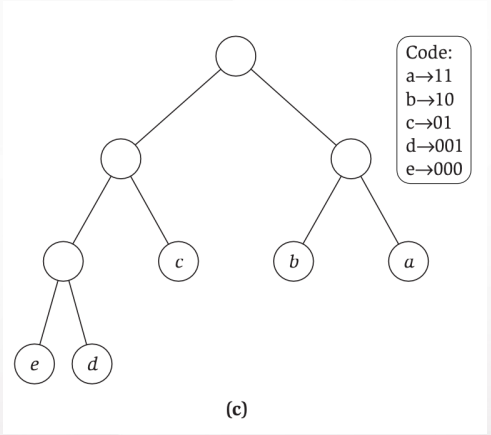
**Representando códigos de prefixo usando árvores binárias**

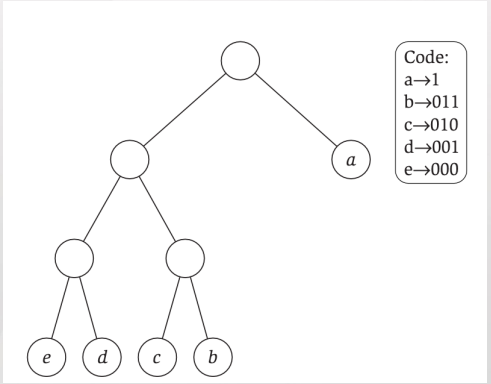
- Assim, estamos procurando a árvore rotulada. que minimiza a média ponderada das profundidades de todas as folhas, onde a média é ponderada pelas frequências das letras que rotulam as folhas:

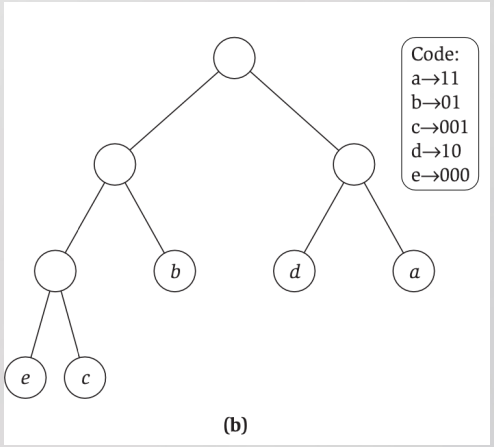
$$ABL(T) = \sum_{x \in S} f_x \, depth_T(x)$$

- Onde $depth_T(x)$ indica a profundidade da letra $x$ na árvore $T$.
- Como primeiro passo para considerar algoritmos para este problema, vamos observar um fato simples sobre a árvore ótima.
- Para isso, precisamos de uma definição:

***Dizemos que uma árvore binária está cheia se cada nó que não é folha tem dois filhos. (Em outras palavras, não há nós com exatamente um filho.)***



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Representando códigos de prefixo usando árvores binárias**

**A.2 A árvore binária correspondente ao código de prefixo ótimo está cheia.**

**Prova**. Isso é fácil de provar usando um argumento de troca.

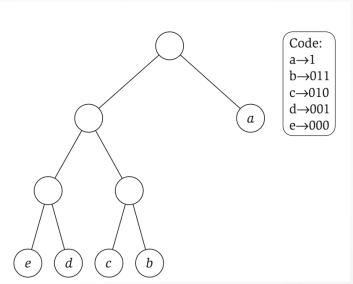
Seja $T$ a árvore binária correspondente ao código de prefixo ótimo e suponha que ela contenha um nó $u$ com exatamente um filho $v$.

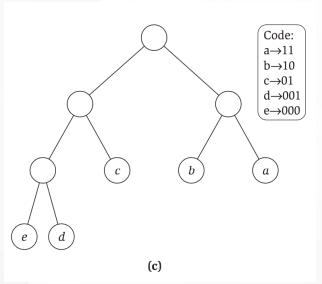
Agora converta $T$ em uma árvore $T'$ substituindo o nó $u$ por $v$.

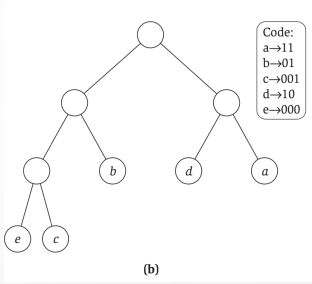
Para ser preciso, precisamos distinguir dois casos:

1. Se $u$ for a raiz da árvore, simplesmente excluímos o nó $u$ e usamos $v$ como raiz.
2. Se $u$ não for a raiz, seja $w$ o pai de $u$ em $T$. Agora excluímos o nó $u$ e tornamos $v$ um filho de $w$ no lugar de $u$.

Essa alteração diminui o número de bits necessários para codificar qualquer folha na sub-árvore com raiz no nó $u$ e não afeta outras folhas.




Assim, o código de prefixo correspondente a $T'$ tem um número médio menor de bits por letra do que o código de prefixo de $T$, contrariando a optimalidade de $T$.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Uma primeira tentativa: a abordagem de cima para baixo**

- Intuitivamente, nosso objetivo é produzir uma árvore binária rotulada na qual as folhas estejam o mais próximo possível da raiz.
- Isso é o que nos dará uma pequena profundidade média da folha.
- Uma maneira natural de fazer isso seria tentar construir uma árvore de cima para baixo, "agrupando" as folhas o mais proximamente possível.
- Portanto, suponha que tentemos dividir o alfabeto $S$ em dois conjuntos $S_1$ e $S_2$, de modo que a frequência total das letras em cada conjunto seja exatamente 0.50.
- Se essa divisão perfeita não for possível, podemos tentar uma divisão que seja tão quase equilibrado possível.
- Em seguida, construímos recursivamente códigos de prefixo para $S_1$ e $S_2$ independentemente e os tornamos as duas sub-árvores da raiz.
- Em termos de cadeias de bits, isso significa colocar um 0 na frente das codificações que produzimos para $S_1$ e colocar um 1 na frente das codificações que produzimos para $S_2$.
- Não está totalmente claro como devemos definir concretamente essa divisão "quase equilibrada" do alfabeto, mas existem maneiras de tornar isso preciso.
- Os esquemas de codificação resultantes são chamados de códigos **Shannon-Fano**, em homenagem a Claude Shannon e Robert Fano, duas das principais figuras iniciais na área da teoria da informação, que lida com a representação e codificação de informações digitais.
- Esses tipos de códigos de prefixo podem ser bastante bons na prática, mas para nossos propósitos atuais eles representam uma espécie de beco sem saída: nenhuma versão dessa estratégia de divisão de cima para baixo garante sempre a produção de um código de prefixo ótimo.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

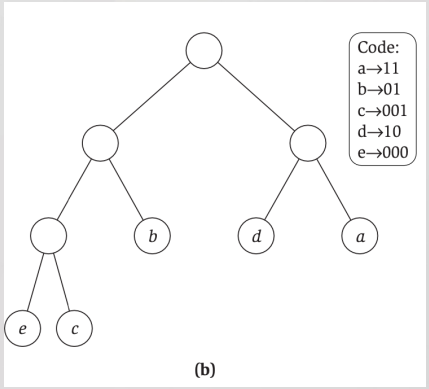
**Uma primeira tentativa: a abordagem de cima para baixo**

- Considere novamente nosso exemplo com o alfabeto de cinco letras $S = \{a, b, c, d, e\}$, e suas frequências são as seguintes:

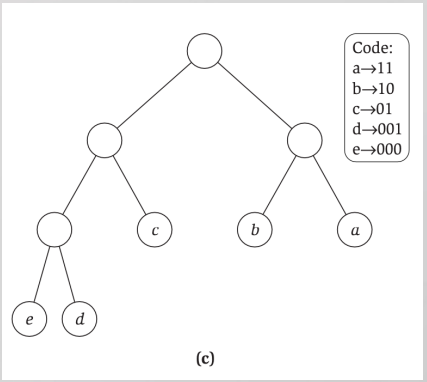
$$f_a = 0.32, f_b = 0.25, f_c = 0.20, f_d = 0.18, f_e = 0.05$$

- Existe uma maneira única de dividir o alfabeto em dois conjuntos de frequência igual: $\{a, d\}$ e $\{b, c, e\}$.
- Para $\{a, d\}$, podemos usar um único bit para codificar cada um.
- Para $\{b, c, e\}$, precisamos continuar recursivamente e, novamente, há uma maneira única de dividir o conjunto em dois subconjuntos de igual frequência.
- O código resultante corresponde ao código $\gamma_1$, dado pela árvore rotulada na Figura (b) e já vimos que $\gamma_1$ não é tão eficiente quanto o código de prefixo $\gamma_2$ correspondente à árvore rotulada na Figura (c).

- Shannon e Fano sabiam que sua abordagem nem sempre produzia o código de prefixo ideal, mas não viam como calcular o código ideal sem pesquisa de força bruta.
- O problema foi resolvido alguns anos depois por **David Huffman**, na época um aluno de pós-graduação que aprendeu sobre a questão em uma aula ministrada por Fano.
- Agora descrevemos as ideias que levaram à abordagem gananciosa que **Huffman** descobriu para produzir códigos de prefixo ótimos.



(b)



(c)

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**E se soubéssemos a estrutura de árvore do código de prefixo ótimo?**

- Uma técnica que costuma ser útil na busca de um algoritmo eficiente é assumir, como um experimento mental, que se conhece algo parcial sobre a solução ótima e, então, ver como se faria uso desse conhecimento parcial para encontrar a solução completa.
- Para o problema atual, é útil perguntar:

***E se alguém nos desse a árvore binária $T^*$ que correspondesse a um código de prefixo ótimo, mas não à rotulagem das folhas?***

- Para completar a solução, precisaríamos descobrir qual letra deveria rotular qual folha de $T^*$ e, então, teríamos nosso código.
- **Quão difícil é isso?**

- Na verdade, isso é bem fácil!
- Começamos formulando o seguinte fato básico.

***A.3 Suponha que $u$ e $v$ sejam folhas de $T^*$, tais que $depth_{T^*}(u) < depth_{T^*}(v)$. Além disso, suponha que em uma rotulagem de $T^*$ correspondente a um código de prefixo ótimo, a folha $u$ é rotulada com $y \in S$ e a folha $v$ é rotulada com $z \in S$. Então $f_y \geq f_z$.***

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

***A.3 Suponha que $u$ e $v$ sejam folhas de $T^*$, tais que $depth_{T^*}(u) < depth_{T^*}(v)$. Além disso, suponha que em uma rotulagem de $T^*$ correspondente a um código de prefixo ótimo, a folha $u$ é rotulada com $y \in S$ e a folha $v$ é rotulada com $z \in S$. Então $f_y \geq f_z$.***

**Prova**. Provamos usando um argumento de troca. Se $f_y < f_z$, então considere o código obtido trocando os rótulos nos nós $u$ e $v$.

Na expressão para o número médio de bits por letra, $ABL(T^*) = \sum_{x \in S} f_x \, depth_T(x)$, o efeito dessa troca é o seguinte: o multiplicador em $f_y$ aumenta (de $\boldsymbol{depth_{T^*}(u)}$ para $\boldsymbol{depth_{T^*}(v)}$), e o multiplicador em $f_z$ diminui em a mesma quantidade (de $\boldsymbol{depth_{T^*}(v)}$ para $\boldsymbol{depth_{T^*}(u)}$).

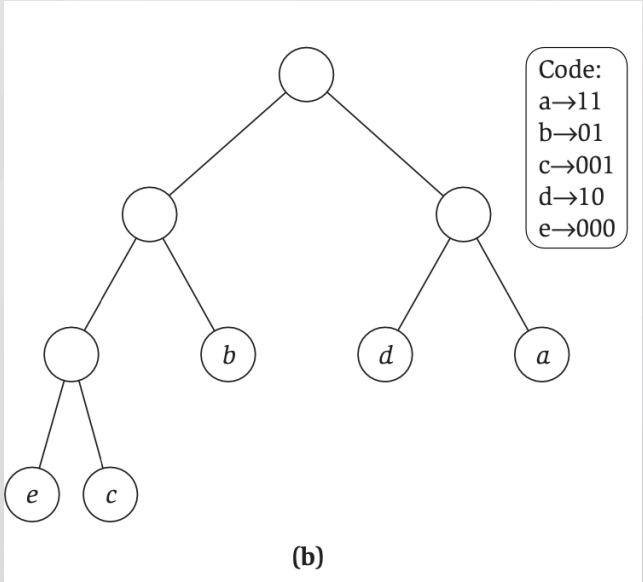
Assim, a alteração na soma geral é $(\boldsymbol{depth_{T^*}(v)} - \boldsymbol{depth_{T^*}(u)})(f_y - f_z)$.

Se $f_y < f_z$, essa mudança é um número negativo, contrariando a suposta optimalidade do código do prefixo que tínhamos antes da troca.

Podemos ver a ideia por trás de **A.3** na Figura (b).

Uma maneira rápida de ver que o código aqui não é ótimo é perceber que ele pode ser melhorado trocando as posições dos rótulos **c** e **d**.

Ter uma letra de frequência mais baixa em uma profundidade estritamente menor do que alguma outra letra de frequência mais alta é precisamente o que exclui para uma solução ótima.



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

***A.3 Suponha que $u$ e $v$ sejam folhas de $T^*$, tais que $depth_{T^*}(u) < depth_{T^*}(v)$. Além disso, suponha que em uma rotulagem de $T^*$ correspondente a um código de prefixo ótimo, a folha $u$ é rotulada com $y \in S$ e a folha $v$ é rotulada com $z \in S$. Então $f_y \geq f_z$.***

- A afirmação ***A.2*** nos dá uma forma, intuitivamente natural e ótimo, de rotular uma árvore $T^*$ proposta:
  1. Primeiro pegamos todas as folhas de profundidade 1 (se houver) e rotule-os com as letras de frequência mais alta em qualquer ordem.
  2. Em seguida, pegamos todas as folhas de profundidade 2 (se houver) e as rotulamos com as próximas letras de frequência mais alta em qualquer ordem.
  3. Continuamos através do folhas em ordem crescente de profundidade, atribuindo letras em ordem decrescente frequência.
- O ponto é que isso não pode levar a uma rotulagem abaixo do ideal de $T^*$ , já que qualquer rotulagem supostamente melhor seria suscetível à troca de **A.3.**
- É fundamental notar que, dentre os rótulos que atribuímos a um bloco de mesma profundidade, não importa qual rótulo atribuímos a qual folha.
- Como as profundidades são todas iguais, os multiplicadores correspondentes na expressão $f_x depth_T(x)$, são iguais e, portanto, a escolha da atribuição entre folhas da mesma profundidade não afeta o número médio de bits por letra.
- **Mas como tudo isso está nos ajudando?**

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

- **Mas como tudo isso está nos ajudando?**
- Não temos a estrutura da árvore ótima $\boldsymbol{T}^*$, e como existem exponencialmente muitas árvores possíveis (do tamanho do alfabeto), não seremos capazes de realizar uma busca de força bruta em todas elas .
- De fato, nosso raciocínio sobre $\boldsymbol{T}^*$ torna-se muito útil se pensarmos não no início desse processo de rotulagem, com as folhas de profundidade mínima, mas bem no final, com as folhas de profundidade máxima - aquelas que recebem as letras com frequência mais baixa.
- Especificamente, considere uma folha $v$ em $\boldsymbol{T}^*$ cuja profundidade seja a maior possível.
- A folha $v$ tem um pai $u$, e por **A.2.** $\boldsymbol{T}^*$ é uma árvore binária completa, então $u$ tem outro filho $w$.
- Referimo-nos a $v$ e $w$ como irmãos, pois eles têm um pai comum. Agora, nós temos

***A.4 $w$ é uma folha de $T^*$***

**Prova.** Se $w$ não fosse uma folha, haveria alguma folha $w'$ na subárvore abaixo dela.

Mas então $w'$ teria uma profundidade maior que a de $v$, contrariando nossa suposição de que $v$ é uma folha de profundidade máxima em $\boldsymbol{T}^*$.

Então $v$ e $w$ são folhas irmãs que estão tão profundas quanto possível em $\boldsymbol{T}^*$.

Assim, nosso processo nível a nível de rotular $\boldsymbol{T}^*$, conforme justificado por **A.3**, chegará ao nível contendo $v$ e $w$ por último.

As folhas neste nível receberão as letras de frequência mais baixa.

Como já argumentamos que a ordem em que atribuímos essas letras às folhas dentro desse nível não importa, existe uma rotulagem ótima na qual $v$ e $w$ obtêm as duas letras de frequência mais baixa de todas.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

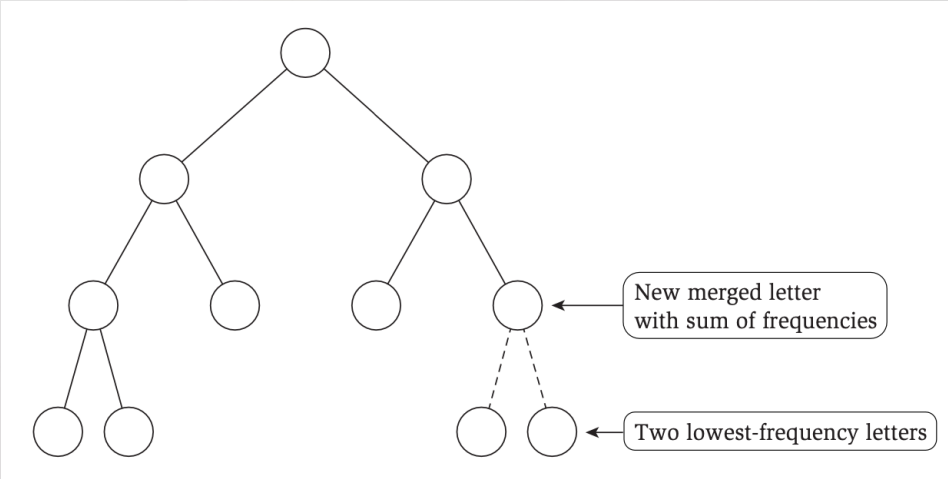
- Combinando o que vimos até agora temos a seguinte afirmação
- ***A.5 Existe um código de prefixo ótimo, com árvore correspondente $T^*$, no qual as duas letras de frequência mais baixa são atribuídas a folhas que são irmãs em $T^*$.***

**Um algoritmo para Huffman Codes**

- Suponha que $y^*$ e $z^*$ sejam as duas letras de frequência mais baixa em $S$.
- Podemos desempatar as frequências arbitrariamente.
- A afirmação **A.5** nos diz algo sobre onde $y^*$ e $z^*$ vão na solução ótima;
- Ele diz que é seguro "prendê-los juntos" ao pensar na solução, porque sabemos que eles acabam como irmãos abaixo de um pai comum.
- Com efeito, esse pai comum age como uma "metaletra" cuja frequência é a soma das frequências de $y^*$ e $z^*$.

- Isso sugere diretamente um algoritmo:
  1. substituímos $y^*$ e $z^*$ por esta meta-letra, obtendo um alfabeto uma letra menor.
  2. Encontramos recorrentemente um código de prefixo para o alfabeto menor e
  3. Em seguida, "abrimos" a meta-letra de volta para $y^*$ e $z^*$ para obter um código de prefixo para $S$.

- Essa estratégia recursiva é representada na Figura.



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Um algoritmo para Huffman Codes**

```
HUFFMAN(S)
1  n = |S|
2  Q = S
3  for i = 1 to n − 1
4       allocate a new node w|
5       y = EXTRACT-MIN(Q)
6       z = EXTRACT-MIN(Q)
7      w|.left = z
8      w|.right = y
9      w|.freq = z.freq + y.freq
10      INSERT(Q w|)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

- O algoritmo HUFFMAN assume $S$ que é um conjunto de $n$ caracters e que cada caractere $x \in S$ é um objeto com um atributo freq ($f_x$) dando sua frequência.
- O algoritmo constrói a árvore $T$ correspondente a um código ótimo de forma bottom-up.
- Ele começa com um conjunto de $|S|$ folhas e executa uma sequência de operações $|S| - 1$ combinações para criar a árvore final.

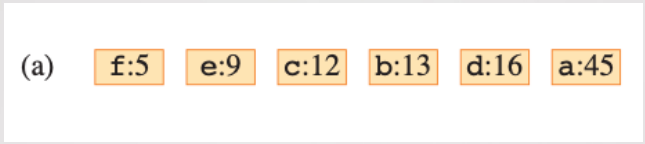
- O algoritmo usa uma priority queue $Q$, com chave no atributo freq. para identificar os dois objetos menos frequentes a serem mesclados.
- O resultado da fusão de dois objetos é um novo objeto ($w$) cuja frequência é a soma das frequências dos dois objetos que foram combinados.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

## Um algoritmo para Huffman Codes - Exemplo

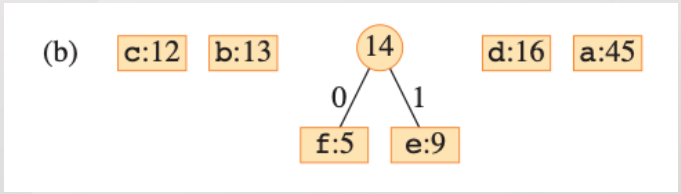
```
HUFFMAN(S)
1  n = |S|
2  Q = S
3  for i = 1 to n − 1
4      allocate a new node w
5      y = EXTRACT-MIN(Q)
6      z = EXTRACT-MIN(Q)
7      w.left = z
8      w.right = y
9      w.freq = z.freq + y.freq
10     INSERT(Q, w)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

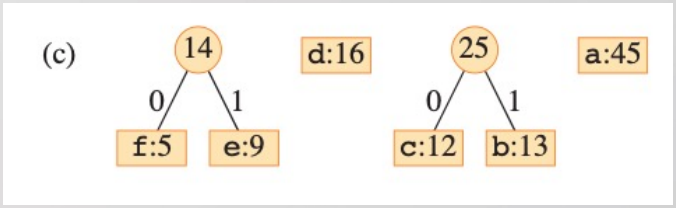
- Para nosso exemplo, o algoritmo de Huffman funciona conforme mostrado na Figura.




- Como o alfabeto contém 6 letras, o tamanho inicial da fila é $n = 6$ e são necessárias $n - 1 = 5$ etapas de combinação para construir a árvore.

- A linha 4 aloca um novo nodo $w$.
- As linhas 4 e 6 armazenam as letras com menor frequência em $y^*$ e $z^*$.
- As linhas 7 e 8 atribuem aos filhos esquerdo e direito de $w$ as letras com menor frequência, ou seja, **f** e **e**.
- A linha 9 calcula a frequência combinada de **f** e **e** e armazena em $w$.freq = 5 + 9 = 14.
- A linha 10 insere $w$ na fila de prioridades.



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

## Um algoritmo para Huffman Codes - Exemplo

```
HUFFMAN(S)
1  n = |S|
2  Q = S
3  for i = 1 to n − 1
4       allocate a new node w|
5       y = EXTRACT-MIN(Q)
6       z = EXTRACT-MIN(Q)
7       w|left = z
8       w|right = y
9       w|freq = z.freq + y.freq
10      INSERT(Q w|)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

(a) f:5 e:9 c:12 b:13 d:16 a:45

(b) c:12 b:13 14 (0: f:5, 1: e:9) d:16 a:45

(c) 14 (0: f:5, 1: e:9) d:16 25 (0: c:12, 1: b:13) a:45

(d) 25 (0: c:12, 1: b:13) 30 (0: 14 (0: f:5, 1: e:9), 1: d:16) a:45

(e) a:45 55 (0: 25 (0: c:12, 1: b:13), 1: 30 (0: 14 (0: f:5, 1: e:9), 1: d:16))

(f) 100 (0: a:45, 1: 55 (0: 25 (0: c:12, 1: b:13), 1: 30 (0: 14 (0: f:5, 1: e:9), 1: d:16)))

- A árvore final representa o código livre de prefixo ideal.

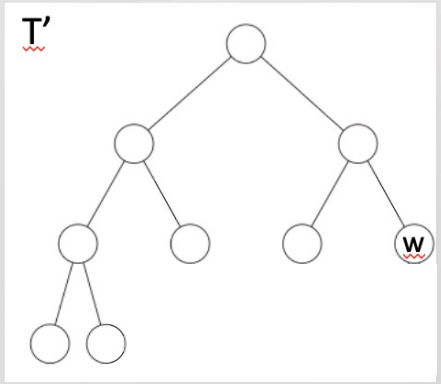
# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

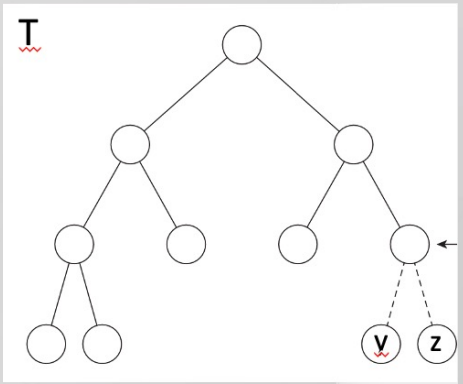
## ANALISANDO O ALGORITMO

```
HUFFMAN(S)
1  n = |S|
2  Q = S
3  for i = 1 to n − 1
4       allocate a new node w|
5       y = EXTRACT-MIN(Q)
6       z = EXTRACT-MIN(Q)
7       w|.left = z
8       w|.right = y
9       w|.freq = z.freq + y.freq
10      INSERT(Q w|)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

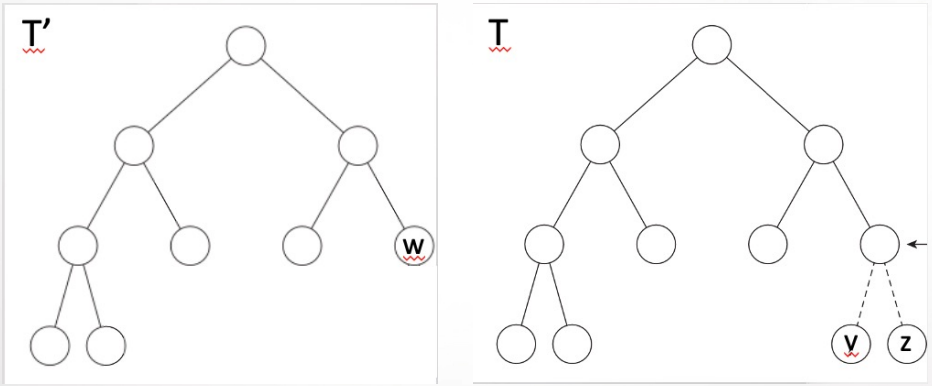
- Primeiro provamos que Algoritmo de Huffman é ótimo.
- Como o algoritmo opera recursivamente, invocando alfabetos cada vez menores, é natural tentar estabelecer a optimalidade por indução no tamanho do alfabeto.
- Claramente a solução é ideal para todos os alfabetos de duas letras (uma vez que usa apenas um bit por letra).
- Portanto, suponha por indução que seja ótimo para todos os alfabetos de tamanho $k-1$ e considere uma instância de entrada que consiste em um alfabeto $S$ de tamanho $k$

- Vamos recapitular rapidamente o comportamento do algoritmo nesta instância.
  - O algoritmo mescla as duas letras de frequência mais baixa $y^*$ e $z^* \in S$ em uma única letra $\omega$.
  - Depois itera no alfabeto menor $S'$ (no qual $y^*$ e $z^*$ são substituídos por $\omega$) e, por indução, produz um código de prefixo ótimo para $S'$, representado por uma árvore binária rotulada $T'$.
  - Em seguida, ele estende isso em uma árvore $T$ para $S$, anexando folhas rotuladas $y^*$ e $z^*$ como filhos do nó em $T'$ rotulado $\omega$.



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)



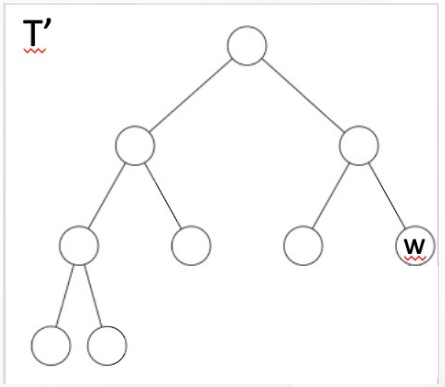

- Existe uma estreita relação entre o número médio de bits usados para codificar letras em S ($ABL(T)$) e o número médio de bits usados para codificar letras em S' ($ABL(T')$), afirmamos que:

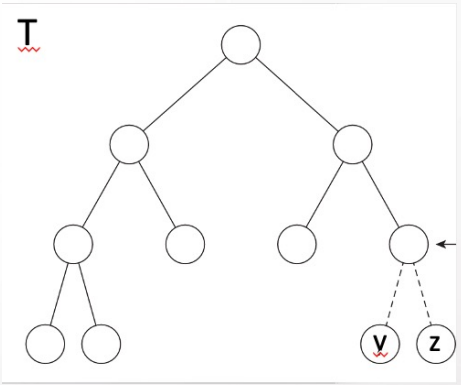
$$\boldsymbol{ABL(T') = ABL(T) - f_w}$$

**Prova**. A profundidade de cada letra $x$ diferente de $y^*$ e $z^*$ é a mesma em $T$ e $T'$.

Além disso, as profundidades de $y^*$ e $z^*$ em $T$ são cada uma maiores que a profundidade de $\omega$ em $T'$.

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)



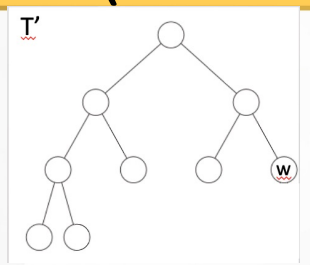

**Prova**. A profundidade de cada letra $x$ diferente de $y^*$ e $z^*$ é a mesma em $T$ e $T'$.

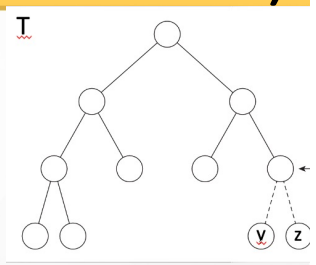
Além disso, as profundidades de $y^*$ e $z^*$ em $T$ são cada uma maiores que a profundidade de $\omega$ em $T'$.

Usando isso, mais o fato de que $f_w = f_{y*} + f_{z*}$, temos:

$$ABL(T) = \sum_{x \in S} f_x \cdot depth_T\,(x)$$

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)




$$ABL(T) = \sum_{x \in S} f_x \cdot depth_T\,(x)$$

$$= f_{y^*} \cdot depth_T(y^*) + f_{z^*} \cdot depth_T(z^*) + \sum_{x \in S \wedge x \neq y^*, z^*} f_x \cdot depth_{T'}(x)$$

$$= \left(f_{y^*} + f_{z^*}\right) \cdot \left(1 + depth_{T'}(w)\right) + \sum_{x \in S \wedge x \neq y^*, z^*} f_x \cdot depth_{T'}(x)$$

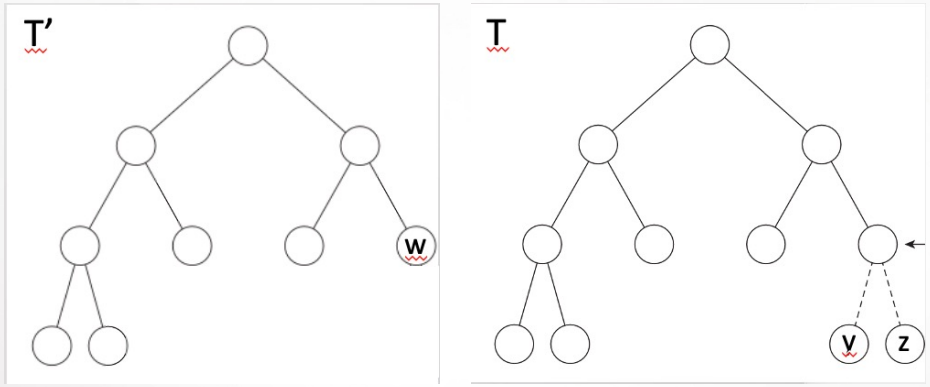
$$= f_w \cdot \left(1 + depth_{T'}(w)\right) + \sum_{x \in S \wedge x \neq y^*, z^*} f_x \cdot depth_{T'}(x) =$$

$$= f_w + f_w \cdot depth_{T'}(w) + \sum_{x \in S \wedge x \neq y^*, z^*} f_x \cdot depth_{T'}(x)$$

$$= f_w + \sum_{x \in S'} f_x \cdot depth_{T'}(x)$$

$$= f_w + ABL(T')$$

# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)




- Usando $ABL(T') = ABL(T) - f_w$, provamos a optimalidade afirmando:

**A.6 O código de Huffman para um determinado alfabeto atinge o número médio mínimo de bits por letra de qualquer código de prefixo.**

**Prova**. Suponha, por contradição, que a árvore $T$ produzida pelo seu algoritmo greedy não seja ideal.

- Isso significa que existe alguma árvore binária rotulada $Z$ tal que $ABL(Z) < ABL(T)$.
- E por **A.5**, existe uma árvore $Z$ na qual as folhas que representam $y^*$ e $z^*$ são irmãs.
- Agora é fácil obter uma contradição, como segue.
- Se excluirmos as folhas rotuladas $y^*$ e $z^*$ de $Z$, e rotularmos seu antigo pai com $\omega$, obteremos uma árvore $Z'$ que define um código de prefixo para $S'$.
- Da mesma forma que $T$ é obtido de $T'$, a árvore $Z$ é obtida de $Z'$ adicionando folhas para $y^*$ e $z^*$ abaixo de $\omega$;
- Assim, a identidade

$$\boldsymbol{ABL(T') = ABL(T) - f_w}$$

também se aplica a $Z$ e $Z'$:

$$\boldsymbol{ABL(Z') = ABL(Z) - f_w}.$$

- Mas assumimos que $ABL(Z) < ABL(T)$ e subtraindo $\boldsymbol{f_w}$ de ambos os lados desta desigualdade obtemos $ABL(Z') < ABL(T')$.
- Isso contradiz a optimalidade de $T'$ como um código de prefixo para $S'$. (Lembre-se no fundo da recursão $T'$ cria uma árvore com apenas duas folhas, e ótima por definição!!!).

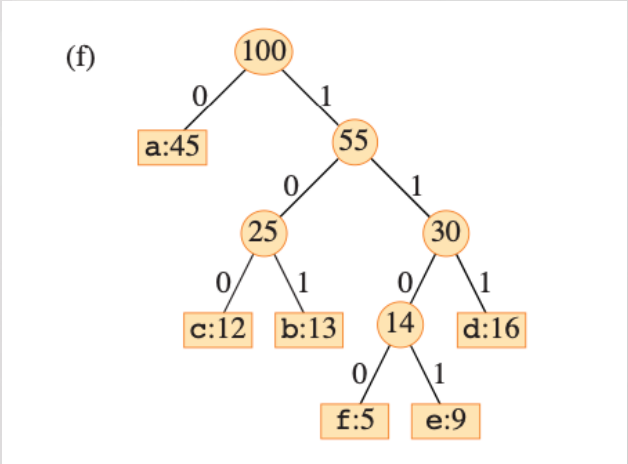
# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

## IMPLEMENTAÇÃO E TEMPO DE EXECUÇÃO:

```
HUFFMAN(S)
1  n = |S|
2  Q = S
3  for i = 1 to n − 1
4       allocate a new node w|
5       y = EXTRACT-MIN(Q)
6       z = EXTRACT-MIN(Q)
7      w|left = z
8      w|.right = y
9      w|freq = z.freq + y.freq
10      INSERT(Q w|)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

- Utilizando uma implementação de filas de prioridade via heaps, podemos fazer cada inserção e extração da execução mínima no tempo $O(\log n)$;
- Somando todas as $n$ iterações, obtemos um tempo total de execução de $O(n \log n)$;

(a) f:5 e:9 c:12 b:13 d:16 a:45



# Algoritmos Gulosos (Greedy): Compressão de dados (Huffman Codes)

**Outros Algoritmos GREEDY importantes**

- Optimal Caching;
- Shortest Paths in a Graph;
- Minimum Spanning Tree;
- ....